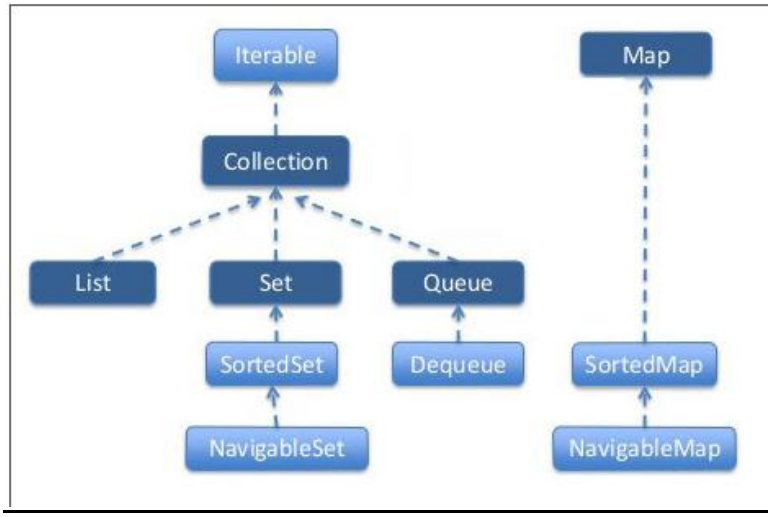
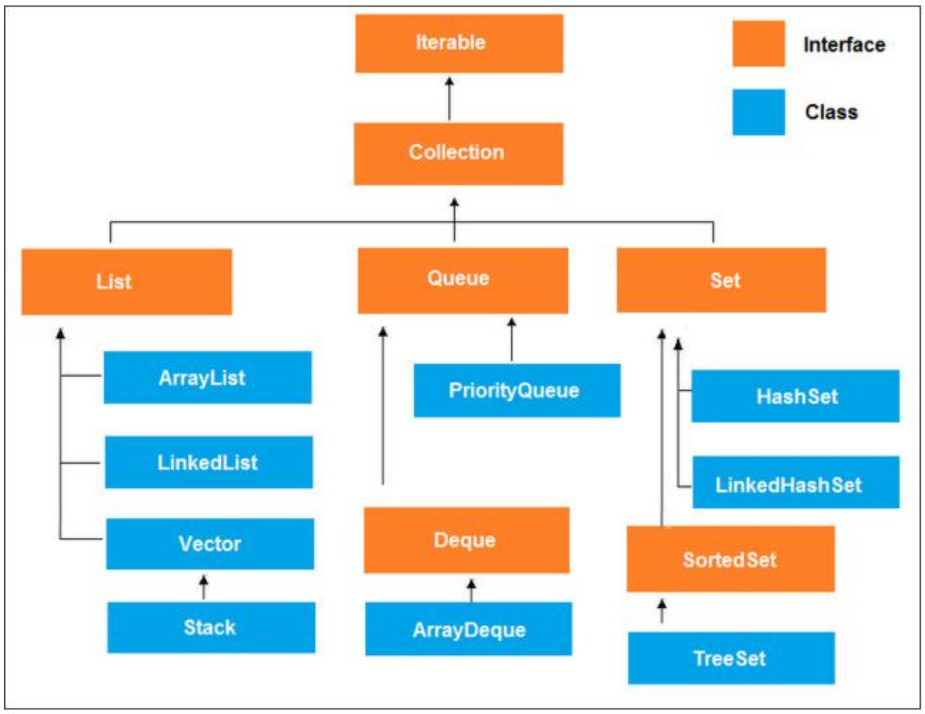
COLECCIONES DE DATOS EN JAVA

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos.

En Java, se emplea la interfaz genérica **Collection** para este propósito. Esto nos permite almacenar cualquier tipo de objeto y usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección…

1. TIPOS DE COLECCIONES





i. **SET**

La interfaz **Set** define una colección que no puede contener elementos duplicados. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**.

ii. **LIST**

La interfaz List define una sucesión de elementos. Sí admite elementos duplicados. A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

- Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
- Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
- Iteración sobre elementos: mejora el Iterator por defecto.

Existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **ArrayList:** esta es la implementación típica. Se basa en un array redimensionable . Es la que mejor rendimiento tiene sobre la mayoría de situaciones.
- **LinkedList**: esta implementación permite que mejore el rendimiento en ciertas ocasiones. Es una lista doblemente enlazada.
  Ninguna de estas implementaciones son sincronizadas.
- **Vector:** El Vector es una implementación similar al ArrayList, con la diferencia de que el Vector si que está sincronizado.

2. METODOS GENERICOS

Todos estos métodos vienen definidos por la interfaz Collection. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

- boolean **add**(Object o)

Añade un elemento (objeto) a la colección. Nos devuelve true si tras añadir el elemento la colección ha cambiado, es decir, el elemento se ha añadido correctamente, o false en caso contrario.

- void **clear()**

Elimina todos los elementos de la colección.

- boolean **contains(Object o)**

Indica si la colección contiene el elemento (objeto) indicado.

- **boolean isEmpty()**

Indica si la colección está vacía (no tiene ningún elemento).

- Iterator **iterator()**

Proporciona un iterador para acceder a los elementos de la colección.

- boolean **remove**(Object o)

Elimina un determinado elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.

- int **size**()

Nos devuelve el número de elementos que contiene la colección.

- Object [] **toArray**()

Nos devuelve la colección de elementos como un array de objetos. Si sabemos de antemano que los objetos de la colección son todos de un determinado tipo (como por ejemplo de tipo String) podremos obtenerlos en un array del tipo adecuado, en lugar de usar un array de objetos genéricos. En este caso NO podremos hacer una conversión cast descendente de array de objetos a array de un tipo más concreto, ya que el array se habrá instanciado simplemente como array de objetos:

```
// Esto no se puede hacer!!!
String [] cadenas = (String []) coleccion.toArray();

String [] cadenas = new String[coleccion.size()];
coleccion.toArray(cadenas);                    // Esto si que funcionará
```

3. METODOS SOBRE LISTAS.

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la **interfaz List**, que además de los métodos generales de las colecciones, nos ofrece los siguientes para trabajar con los índices:

- **void add(int indice, Object obj)**

Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

- **Object get(int indice)**

Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

- **int indexOf(Object obj)**

Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve - 1 si el objeto no se encuentra en la lista.

- **Object remove(int indice)**

Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

- **Object set(int indice, Object obj)**

Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha.

4. COMPARACION DE OBJETOS. SOBRECARGA DE EQUALS

Cuando se sobrecarga el método equals se deben cumplir las siguientes propiedades:

- Reflexividad: x.equals(x) devuelve siempre verdadero, si no es nulo.
- Simetría: para cualquier par de instancias no nulas, x.equals(y) devuelve verdadero si y sólo si y.equals(x) también devuelve verdadero.

- Transitividad: si x.equals(y)==true y y.equals(z)==true, entonces x.equals(z) también será verdadero, para cualesquiera instancias no nulas.
- Consistencia: múltiples llamadas al método con las mismas instancias devuelven el mismo resultado.
- Comparación con null falsa: x.equals(null) devuelve falso.

i. **Interfaz Comparable**

Hay algoritmos, como Collections.sort( ), que requieren que los objetos tengan un método compareTo() que devuelva un número negativo, positivo o cero, según si un objeto es menor que el otro, mayor, o igual.

Este método no viene en Object para poder sobrecargarlo, sino en la interfaz Comparable que tenemos que implementar, y que nos obligará a implementar también el método compareTo.

Ejemplo

Realiar una lista de personas. El programa nos dirá la persona con mayor edad y luego nos mostrará la línea ordenada

//CLASE PERSONA//

```
package ejemplocomparable;

public class Persona implements Comparable<Persona>{

public int dni, edad;

    public Persona( int d, int e){

        this.dni = d;

        this.edad = e;

}
@Override
      public int compareTo(Persona o)
    {
        int res=0;
```

```
        if (this.edad<o.edad)
        {res=-1;  }
        else if (this.edad>o.edad)
        {res=1;}
        else
        { if(this.dni>o.dni)
              res=1;
           else
              res=-1;
        }
        return res;

     }

   public String toString() {
      return "Persona{" + "dni=" + dni + ", edad=" + edad + '}';
   }

}
//PROGRAMA MAIN IMPLEMETADO CON UN ARRAYLIST Y
//UTILIZANDO EL SORT.
package ejemplocomparable;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class EjemploComparable {

   public static void main(String[] args) {
       ArrayList <Persona> a= new ArrayList<Persona>();
      // TODO code application logic here
      for (int i=0; i<10; i++)
         a.add(new Persona(i, (int)(Math.random()*10)));
      System.out.println("la persona mayor tiene el dni " +
mayor(a).dni + "sus edad es" + mayor(a).edad);
      Collections.sort(a);
      for(Persona x:a)
      {
```

```
            System.out.println( x.toString());
        }
    }

    static Persona mayor(ArrayList <Persona> a)
    {   Persona y=new Persona(-1,-1);

        for (Persona x: a)
        {  if((x.compareTo(y)==1))
            y=x;

        }
        return y;
    }

}
```